AUTOMATIC PARALLELIZATION OF LOOPS

by

B. Muralidhara Rao

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

FEBRUARY, 1995

AUTOMATIC PARALLELIZATION OF LOOPS

[illegible]

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of

MASTER OF TECHNOLOGY

by

B. Muralidhara Rao

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

February, 1995

CSE-1995-M-RAO-AUT

Certificate

This is to certify that the work contained in the thesis titled **Automatic Parallelization of Loops** by **B. Muralidhara Rao (Roll No: 9311104)**, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

[illegible]
Feb 28, 1995

Dr. Sanjeev Kumar Aggarwal
Associate Professor,
Dept. of Computer Science & Engg.,
IIT, Kanpur

Acknowledgements

My foremost gratitude to my guide, Dr. Sanjeev Kumar for his constant encouragement and excellent suggestions during my thesis work.

Sandeep's suggestions in moulding the thesis to a presentable form is of enormous help.

My frequent discussions with Sandeep and Govindraj helped me clearing many of the concepts in this exiting domain of parallel compilers.

Lastly but not the least, I thank Hari Babu, Raja Babu, Ram kumar Shankar, Pawan, AdiNarayana, Mantri and Radha Krishna for making my stay at IITK a memorable one.

Abstract

This work focuses on two major aspects of a parallelizing compiler for Fortran-D: data dependence analysis and loop restructuring. The traditional approach of data flow analysis, employed by compilers for sequential machines, is not sufficient for exploiting the potential parallelism present in the loops. The concept of data dependence analysis captures the reference pattern of the arrays in the loops. FRAMES, the earlier version of the Fortran-D compiler, developed by our group, relied on two primitive tests, namely, the GCD and Banerjee's tests, for data dependence analysis. These tests are highly conservative in nature and hence, fail to extract the full amount of parallelism present in the scientific programs. More sophisticated dependence tests are implemented, which efficiently tackle coupled subscripts, trapezoidal regions and symbolic variables. A neat interface to data dependence analysis is also provided. This interface enables programmers to incorporate new dependence tests without difficulty.

The loop restructuring deals with transforming the loop-nests to map onto the underlying architecture. The validity of these transformations is ensured by preserving the data dependence relations provided by the data dependence analysis. Some sophisticated loop restructuring techniques: loop interchange, cycle shrinking, loop distribution and loop fusion, are implemented. These restructuring techniques significantly enhance the parallelizing capability of FRAMES.

This thesis discusses the design and implementation issues involved in the above mentioned aspects of FRAMES.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

The speed of sequential machines has approached its theoretical limits and parallel computers seems to be the only way to increase the computational power. Parallel computers, however, are not useful unless they are easily programmable. There are two aspects that discourage scientists to use any parallel machine. Firstly, there is no programming language available that enables scientists to easily write parallel programs. The second aspect is the lack of an automatic restructuring compiler to parallelize already existing billions of lines of sequential code. The project FRAMES at IIT, Kanpur addresses the second issue mentioned above. FRAMES[15, 14, 5, 9] is a restructuring compiler, which converts code written in Fortran-77 into Fortran-D [7], an extension of Fortran for MIMD architecture.

1.2 Structure of FRAMES

There are four main components of FRAMES.

1. Front end.

2. Data dependence analyzer.

3. Restructurer.

4. Scheduler.

1.2.1 Front end

The Front end parses the input program, performs interprocedural analysis and finally, applies various optimizations. The parser generates the abstract syntax tree (AST) corresponding to the input program. The AST is used for interprocedural analysis and scalar optimizations. Some of the scalar optimizations that Front end performs are copy propagation and constant folding[1]. The various optimizations also include loop normalization which is very useful for the later phases.

1.2.2 Data dependence analyzer

Traditionally, the compilers treat complex variables, such as arrays, in a very conservative way. They assume a reference to an element of a complex variable as a reference to the entire data object. The scientific programs spend most of their execution time in the loops which contain array references[12]. Obviously, the traditional data flow analysis leaves most of the parallelism in the loops unexploited. The data dependence analysis used in a restructuring compiler, determines whether a given pair of array references results in a dependence. There are number of data dependence tests available which use the subscript expressions and loop bounds to solve the dependence problem. General literature on this subject is widely available[18, 13, 10, 3]. A data dependence graph (DDG) is constructed based upon the data dependence decision algorithms to make the dependence relations between statements explicit.

1.2.3 Restructurer

The various loop restructuring techniques are applied on the DDG [17, 12]. The data structure needed for program representation is program dependence graph(PDG)[4] which supports operations on the program very efficiently. The restructurer transforms the loop-nests in such a way that they can be executed on different processors to increase the speed-up. Some of the loop restructuring techniques that are implemented are node-splitting, loop interchanging, loop fusion, loop distribution and cycle shrinking.

1.2.4 Scheduler

The loop restructuring transforms the loop-nests such that they can be executed on different processors. The scheduler[12] assigns processors to loop iterations, so as to maximally utilize the available processors. The main goal of scheduling is load balancing and synchronizing each loop iteration. The two types of scheduling used are static and dynamic. The static scheduling assigns various iterations to different processors at the time of compilation. Sometimes, the static scheduling cannot be done owing to insufficient information available at the time of compilation. For example, the loop bounds may not be known or the loop body may have conditional statements. In such cases, the dynamic scheduling is employed. It inserts code in the user programs to assign iterations to various processors during execution.

1.3 Objectives of the thesis

The main objective of the work, reported in this thesis, is to enhance the data dependence analysis and restructuring capabilities of FRAMES. Tests such as GCD and Banerjee were implemented earlier. These tests are less powerful and cannot handle the whole spectra of types of subscript expressions that are usually encountered in programs. One of the objectives of the thesis is to add various dependence tests to cover all types of subscript expressions. In restructurer, node splitting was the only restructuring technique that was implemented earlier. Incorporation of some more general restructuring techniques into FRAMES is one more objective of the thesis.

1.4 Organization of the thesis

- Chapter 2 discusses the basic concepts of data dependence analysis. Some of the algorithms that are used during implementation are also discussed.

- Chapter 3 discusses different aspects of dependence tests. It explains various tests, Banerjee's test, Lambda test, Power test and Omega test, in detail.
- Chapter 4 gives a brief description of PDG. Algorithms for various restructuring techniques(loop interchange, cycle shrinking, loop distribution) are given in this chapter.
- Chapter 5 is dedicated to the testing of various aspects of code, possible future developments, short commings and conclusions.

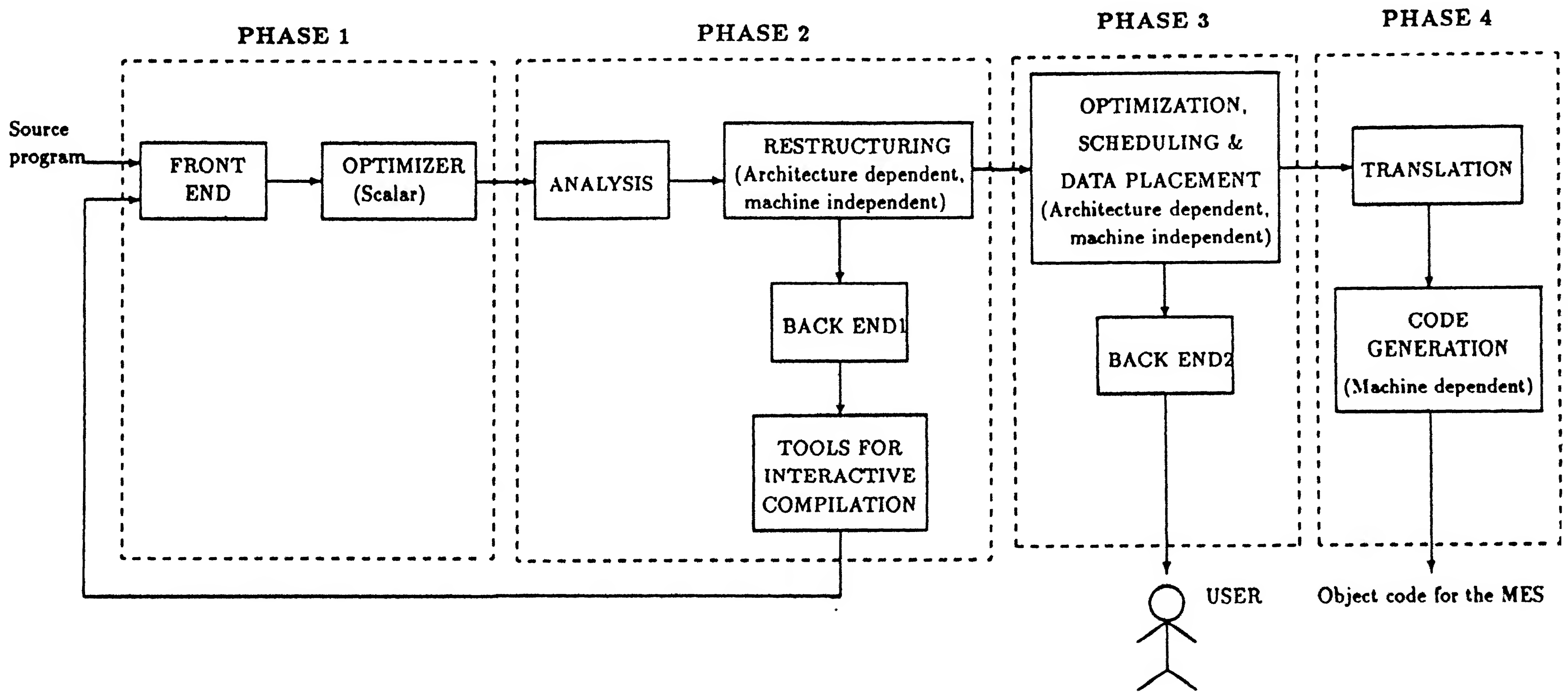
PHASE 1
PHASE 2
PHASE 3
PHASE 4
Source program
FRONT END
OPTIMIZER (Scalar)
ANALYSIS
RESTRUCTURING (Architecture dependent, machine independent)
BACK END1
TOOLS FOR INTERACTIVE COMPILATION
OPTIMIZATION, SCHEDULING & DATA PLACEMENT (Architecture dependent, machine independent)
BACK END2
USER
TRANSLATION
CODE GENERATION (Machine dependent)
Object code for the MES

Figure 1.1: The structure of the compiler

Chapter 2

Data Dependence Analysis

2.1 Introduction

The dependences between two statements in a given program can arise in two different ways. Firstly, when the value computed in one statement S_1 is dependent on some other statement S_2 and hence, the computed value would be incorrect if the order of execution of the two statements is reversed. Such dependences are called *data dependences.* In the the following example, the statement S_2 depends upon S_1 since the value assigned to W in S_2 depends on X which is computed in S_1. If the order of execution is reversed, the value of W may be incorrect.

```
S1:   X = Y*Z
S2:   W = (X + 1)*V
```

The second type of dependences, called *control dependences*, occur between a predicate and a statement such that the value of the predicate immediately controls the execution of the statement. Consider the following example.

```
S1:   if (X) then
S2:          Y = Y + 1
```

Here, S_2 depends upon the predicate X in statement S_1, since the value of X determines whether S_2 is executed or not. The control dependences have been thoroughly discussed in [1, 4].

Traditional data flow analysis analyzes dependences involving scalar variables and treats the complex variables similarly. That is, it is assumed that any two references to the complex variable access the whole data object. This conservative approach is, obviously, not sufficient for restructuring compilers because it leaves a great amount of parallelism unexploited. The operations on an array may be performed in parallel if the various references to the array access different locations. Essentially there exists three types of parallelism in programs.

Coarse grain parallelism is at the subroutine level. Usually, the computation is organized into subroutines or coroutines. Various independent subroutines can be executed in parallel.

Medium grain parallelism exists at the loop level. Several different types of parallel loops exists depending on the kind of dependence graph of the loop-nest.

Fine grain parallelism is achieved when several independent basic blocks can be executed in parallel. Fine grain parallelism also includes parallelism at the statement and operation level in the given program.

The Data dependence analysis is intended to address medium and fine grain parallelism.

2.2 Data Dependence Concepts

Consider the following general form of loop-nest which represents perfectly nested as well as imperfectly nested loops.

```
DO $I_1$ = $L_1$, $U_1$
  ⋮
  DO $I_s$ = $L_s(i_1, i_2, \ldots, i_{s-1})$, $U_s(i_1, i_2, \ldots, i_{s-1})$
    DO $I_{s+1}$ = $L_{s+1}(i_1, i_2, \ldots, i_s)$, $U_{s+1}(i_1, i_2, \ldots, i_s)$
      ⋮
      DO $I_p$ = $L_p(i_1, i_2, \ldots, i_{p-1})$, $U_p(i_1, i_2, \ldots, i_{p-1})$
        $S_1$:  $A[f_1(i_1, i_2, \ldots, i_p), f_2(i_1, i_2, \ldots, i_p), \ldots, f_m(i_1, i_2, \ldots, i_p)]$
      ENDDO
      ⋮
    ENDDO
    DO $I_{p+1}$ = $L_{p+1}(i_1, i_2, \ldots, i_s)$, $U_{p+1}(i_1, i_2, \ldots, i_s)$
      ⋮
      DO $I_q$ = $L_q(i_1, i_2, \ldots, i_{q-1})$, $U_q(i_1, i_2, \ldots, i_{q-1})$
        $S_2$:  $F(A[g_1(i_1, i_2, \ldots, i_q), g_2(i_1, i_2, \ldots, i_q), \ldots, g_m(i_1, i_2, \ldots, i_q)])$
      ENDDO
      ⋮
    ENDDO
  ENDDO
  ⋮
ENDDO
```

where

- f_i and g_i, $(1 \le i \le m)$, are arbitrary subscript expressions.
- m is the number of dimensions of the array reference.
- F is an expression involving an array reference.

S_2 depends on S_1 if and only if there exists two integer vectors $\vec{i}(i_1, \ldots, i_s, i_{s+1}, \ldots, i_p)$ and $\vec{j}(j_1, \ldots, j_s, j_{p+1}, \ldots, j_q)$ such that $L_k \le i_k, j_k \le U_k$, $(1 \le k \le q)$ and the following system of equations is satisfied.

$$\begin{aligned} f_1(i_1, i_2, \ldots, i_p) &= g_1(j_1, j_2, \ldots, j_q) \\ f_2(i_1, i_2, \ldots, i_p) &= g_2(j_1, j_2, \ldots, j_q) \\ &\vdots \\ f_m(i_1, i_2, \ldots, i_p) &= g_m(j_1, j_2, \ldots, j_q) \end{aligned} \tag{2.1}$$

Intuitively, a dependence between the statements S_1 and S_2 means that statement S_1 computes a value in an instance $\vec{i}$ that is subsequently used by the statement S_2 in the instance $\vec{j}$. If f_i's and g_i's are permitted to be arbitrary functions of the loop index variables, then solving the data dependence problem becomes extremely difficult. When f_i's and g_i's are restricted to linear functions, the problem becomes more tractable. It should be emphasized, however, that the simplified problem is in the class of NP-complete problems. Assuming that the functions f_i's and g_i's are linear functions of the loop index variables, the dependence problem reduces to finding simultaneous solutions to the equations of the form

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n = c \tag{2.2}$$

Such equations are known as linear diophantine equations. Some properties of the linear diophantine equations are discussed in Section 2.4.

2.2.1 Types of Dependences

There are three types of dependences that can exist between two given statements S_1 and S_2 in a program. Assume that the control flow within a program can reach S_2 after passing through S_1. Let $IN(S)$ be the set of memory locations read in statement S and $OUT(S)$ be the set of memory locations written in statement S.

Definition 2.1 Flow Dependence: *A flow dependence, denoted by $S_1\delta S_2$, exists between S_2 and S_1 iff $OUT(S_1) \cap IN(S_2) \neq \emptyset$. The following example illustrates the relation $S_1\delta S_2$.*

```
S1:   X = ...
S2:   ... = X
```

Definition 2.2 Anti Dependence: *If $IN(S_1) \cap OUT(S_2) \neq \emptyset$, then there exists anti dependence, written $S_1 \bar{\delta} S_2$, between the two statements. In the following case, $S_1 \bar{\delta} S_2$ holds.*

```
S1:   ... = X
S2:   X = ...
```

Definition 2.3 Output Dependence: *S_1 and S_2 are involved in output dependence if $OUT(S_1) \cap OUT(S_2) \neq \emptyset$. The notation $S_1 \delta^o S_2$ is used to describe the output dependence between the statements S_1 and S_2. In the example given below, S_1 and S_2 both assign a value to X and hence, are involved in output dependence.*

```
S1:   X = ...
S2:   X = ...
```

Anti and output dependences are false dependences and can be eliminated by simple techniques such as variable renaming. Flow dependence is also referred to as *true dependence* in the literature and it cannot be eliminated.

2.2.2 Direction and Distance Vectors

The dependence between two statements can be characterized using the distance and direction vectors. Using these vectors, we can determine

- loops that carry the dependence
- direction of the dependence
- the number of iterations of a loop between a pair of references involved in a dependence.

Definition 2.4 A direction vector is a s-tuple $\Psi(\Psi_1, \Psi_2, \ldots, \Psi s)$, where $\Psi_k \in \{<, =, >, *\}$ and we write $S_v \delta_{(\Psi_1, \Psi_2, \ldots, \Psi_s)} S_w$ when

- there exist particular instances of S_v and S_w, say $S_v[i_1, i_2, \ldots, i_s]$ and $S_w[j_1, j_2, \ldots, j_s]$, such that $S_v[i_1, i_2, \ldots, i_s] \delta S_w[j_1, j_2, \ldots, j_s]$ and

- $\theta(i_k)\Psi_k\theta(j_k)$ for $1 \leq s$.

A direction vector has s elements corresponding to the common loops enclosing both the references. ψ_i = ' >' signifies that the dependence is carried in backward direction with respect loop i. Similarly, ψ_i = ' <' means that the dependence is carried in the forward direction for loop i. Finally, an '=' element in the direction vector denotes that the dependence is loop independent. If all the directions are valid, or if the direction of the dependence is unknown then the corresponding element is usually represented by '*'.

Definition 2.5 A distance vector, represented by $D(d_1, d_2, \ldots, d_s)$ where d_i is an integer and we say $S_v\delta_{(d_1,d_2,\ldots,d_s)}S_w$ when

- there exist particular instances of S_v and S_w, say $S_v[i_1, i_2, \ldots, i_s]$ and $S_w[j_1, j_2, \ldots, j_s]$, such that $S_v[i_1, i_2, \ldots, i_s]\delta S_w[j_1, j_2, \ldots, j_s]$

- $j_k = i_k + d_k$ and $1 \leq k \leq s$.

The direction vector can be easily obtained from the distance vector by considering the sign of an element in distance vector. The positive and negative elements of the distance vector respectively correspond to ' <' and ' >' in the direction vector. Similarly, if an element in the distance vector is 0, it implies '=' for the corresponding element of the direction vector.

The depth of dependence denotes the loop due to which the data dependence between two statements arises. The depth of dependence is defined as follows.

Definition 2.6 S_2 depends on S_1 at depth d (denoted $S_1\Delta_d S_2$), if there exists a $k \geq d$ such that $S_1\delta_k S_2$. In other words,

$$\Delta_d = \sum_{k=d}^{\infty} \delta_k$$

In the following example, the direction vector is $\{<, >\}$ and the distance vector is $\{1, -2\}$. The depth of dependence is 1.

```
DO I = 1, N
  DO J = 1, M
    A(I, J) = ...
    ...     = A(I - 1, J + 2)
  ENDDO
ENDDO
```

2.2.3 Loop carried and Loop independent Dependence

There are two ways in which data dependence can arise between different statements in a loop-nest. The value stored by one statement may be fetched by another statement in a later or in the same iteration of the loop. The former case is known as loop carried dependence while the other is called loop independent dependence.

Definition 2.7 S_2 has a loop carried dependence on S_1 if there exists i_1 and i_2 such that $1 \leq i_1 < i_2 \leq N$ and $f_r(i_1) = g_r(i_2)$.

Definition 2.8 S_2 has a loop independent dependence on S_1 if there exists i $(1 \leq i \leq N)$ such that $S_2 > S_1$ and $f_r(i) = g_r(i)$.

For the sake of clarity, the above definitions are given with respect to a single loop. Our interest lies mainly in loop carried dependence, because in MIMD architectures the entire loop body for a particular loop iteration is executed on a single processor. Since all the statements within the loop body are executed without any possible change in the execution order, loop independent dependences are always preserved.

2.3 Data Dependence Graph

A data dependence graph is a conceptual representation of inter-statement dependences in a loop-nest. Each statement is represented by a vertex and each dependence is denoted by a directed edge in the graph.

```
DO I = 1, 100
   $S_1$: A (I) = 5.0
   DO I = 1, 100
      $S_2$ : B(I, J) = A(I+1)
   ENDDO
   DO J = 1, 100
      $S_3$: C(I, J) = B(I, J)
      $S_4$ : A(I+1) = C(I-1, J+1)
   ENDDO
ENDDO
```

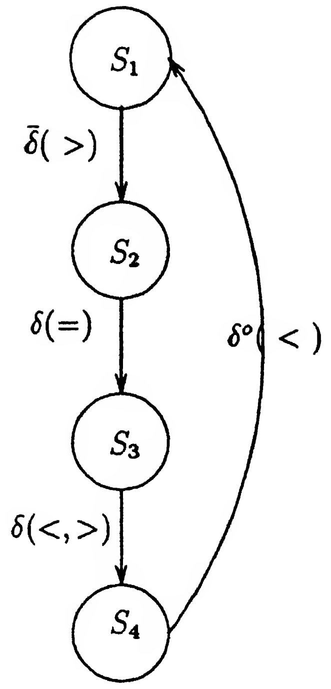

Figure 2.1: Loop nest and its corresponding DDG

Definition 2.9 A dependence graph G is an ordered pair (V, E) where V is the set of vertices representing the statements in the given loop-nest and E is the set of edges representing inter-statement dependences. Each edge $e \in E$ may be viewed as quadruple $< S_1, S_2, t, v >$, where there is a dependence, of type t (flow, anti, output), from S_1 to S_2 and v is either a direction or distance vector associated with the dependence.

The Figure 2.1 contains a loop-nest and the corresponding DDG.

Definition 2.10 For two statements S_1 and S_2, $\eta^0(S_1, S_2)$, the nesting level of the direct dependence of S_2 on S_1 is the maximum depth at which the dependence exists, that is

$$\eta^0(S_1, S_2) = \begin{cases} max(k \geq 1 | S_1 \delta_k S_2), & if S_1 \Delta S_2 \\ 0, & otherwise \end{cases}$$

2.4 Linear Diophantine Equations

A linear diophantine equation is of the form given in Equation 2.2. The linear diophantine equations play a significant role in data dependence analysis. Since the subscript expressions used in practice are linear functions of the loop index variable, the dependence between a pair of array references can be formulated by a system of linear diophantine equations. The following theorems describe an elementary result from number theory which can be used to determine whether there is an integer solution to a given linear diophantine equation. An extension to a system of linear diophantine equations is also presented.

Theorem 2.1 *Let $a_1, a_2, \ldots, a_n$, and c denote integers such that a_i's are all not 0, and let $d = \gcd(a_1, a_2, \ldots, a_n)$. The Equation 2.2 has an integer solution if and only if d divides c.*

Theorem 2.2 *Assume that d divides c. Let $c' = c/d$, $\mathbf{A} = (a_1, a_2, \ldots, a_n)t$, $\mathbf{D} = (d, 0, \ldots, 0)t$, and $\mathbf{U}$ any $n \times n$ unimodular integer matrix satisfying $\mathbf{UA} = \mathbf{D}$. The general solution to Equation 2.2 is then given by the formula*

$$(x_1, x_2, \ldots, x_n) = (c', t_2, t_3, \ldots, t_n).\mathbf{U}$$

where t_2, t_3, ..., t_n are arbitrary integers.

Theorem 2.3 *Consider a system of m linear diophantine equations in n variables:*

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= c_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= c_2 \\ \vdots \qquad\qquad & \quad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n &= c_m \end{aligned} \tag{2.3}$$

The above equations can be written in the matrix notation as follows.

$$\mathbf{xA} = \mathbf{C}$$

where $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, $\mathbf{C} = (c_1, c_2, \ldots, c_m)^T$, *and*

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{bmatrix}$$

where a_{ij}'s and c_k's are integer constants. Let $\mathbf{U}$ *denote an $n \times n$ unimodular integer matrix and* $\mathbf{D}$*, an $n \times n$ echelon integer matrix such that* $\mathbf{UA} = \mathbf{D}$*. If an $n \times 1$ integer matrix* $\mathbf{t}$ *exists satisfying* $\mathbf{tD} = \mathbf{C}$*, then* $\mathbf{x} = \mathbf{tU}$ *is a solution to the system. Conversely, if* $\mathbf{x}$ *is a solution, there must exists an $n \times 1$ integer matrix* $\mathbf{t}$ *satisfying*

$$\mathbf{tD} = \mathbf{C} \quad \textit{and} \quad \mathbf{x} = \mathbf{tU}$$

These three theorems determine the existence of integer solutions to a given set of linear diophantine equations and used extensively in data dependence analysis. The interested reader may refer to [3] for more details and proofs of the above theorems.

2.5 Data Dependence Frame Work

A generalized frame work for computing direction vectors was proposed by Wolfe [17]. It determines whether and under what conditions the array regions accessed by the two references intersect. The two regions share common elements when the subscript functions in Equation 2.3 have a simultaneous solution.

First, a test is made to determine the possible existence of dependence for the direction vector $(*, *, \ldots, *)$. If dependence is not ruled out, then one '$*$' direction vector element is refined to '$<$', '$=$' or '$>$'. If dependence is not ruled out with this refined direction vector, then the regions accessed by the two references are disjoint. This way dependence testing is done on a hierarchy. The hierarchy for two loops is shown in Figure 2.2.

$$\begin{tikzcd}
 & & & & (*,*) \arrow[dlll] \arrow[d] \arrow[drrr] & & & & \\
 & (*,<) \arrow[dl] \arrow[d] \arrow[dr] & & & (*,=) \arrow[dl] \arrow[d] \arrow[dr] & & & (*,>) \arrow[dl] \arrow[d] \arrow[dr] & \\
(<,<) & (<,=) & (<,>) & (<,=) & (=,=) & (>,=) & (<,>) & (=,>) & (>,>)
\end{tikzcd}$$

Figure 2.2: Refinement of direction vector

2.5.1 Legal Direction Vectors

The validity of direction vectors also depends upon the control dependences of the references. Assume that S_1 and S_2 are two statements involved in a dependence. In the following example, the two statements S_1 and S_2 are executed under same control conditions. Hence all the direction vectors are valid.

```
DO I = L, U
    $S_1$ : ...
    $S_2$ : ...
ENDDO
```

On the other hand, the conditional statements in the loop may change the possible direction vectors. Consider the following example.

```
DO I = L, U
   IF (...) THEN
      $S_1$ : ...
   ELSE
      $S_2$ : ...
   ENDIF
ENDDO
```

The two statements S_1 and S_2 are executed under mutually exclusive conditions. Therefore, none of the dependence directions is valid. A loop exit, as illustrated in the following example, can also affect the validity of some of the dependence directions.

```
DO I = L, U
   $S_1$ : ...
   IF (...) THEN
      ...
      $S_2$ : ...
      goto label
   ENDIF
ENDDO
label:
```

Whenever S_2 is executed, the unconditional exit from the loop guarantees that S_1 would never be executed after S_1. Hence, $S_1 \delta_= S_2$ is allowed but $S_2 \delta_> S_1$ is no longer possible. For an inner loop in a loop-nest, the rules are same as for single loop when direction vector elements for outer loops are all $(=)$. When the direction vector for any outer loop is $(<)$, then any direction for an inner loop is allowed.

2.6 Implementation Details

In this section some of the implementation issues are considered. Data dependence algorithm is given below. The naming of the functions and identifiers in the code implicitly specifies their purpose.

Algorithm: Data Dependence Analysis

```
Input: Loopheader of the loop-nest in question.
Output: Data dependence graph.
DDA()
{
   get loop limits in the loop-nest;
   initialize the data dependence graph with vertices;
   for ($stat_1$ = each vertex in the graph )
      for ($stat_2$ = each vertex in the graph starting from $stat_1$)
         if ((any of the array defined in both the statements) or
            (any of the array defined in one statement is used in
            another statement)) then {
            get arrays referenced in $stat_1$ and $stat_2$;
            for (each array reference in $stat_1$)
               for (each array reference in $stat_2$) {
                  $ref_1$ = write reference among the two;
                  $ref_2$ = remaining reference in the pair;
                  standardize subscripts in references $ref_1$ and $ref_2$;
                  call dependence tests;
                  enter the results into log file for analysis purpose;
                  if (all the tests report dependence) {
                     get the intersection of sets of direction vectors
                        reported by the tests;
                     get the best distance vector;
                     find type of dependence;
                     add corresponding edges into the DDG;
                  }
               }
         }
}
```

Each linear expression is stored in the form of a vector of fixed length. The coefficient of i^{th} variable in the expression is the i^{th} element of the vector. The constant term is the 0^{th} element. `get_loopinfo()` routine analyzes each loop in the given loop-nest. Each loop is named by a unique number. The loop limits expressions are saved in `upperb` and `lowerb`. The loop limits can be

- constant
- linear equation in terms of outer loop index variables
- non-linear containing unknown symbolic variables

Two vectors are used to save the nature of each loop limits. The corresponding element of the loop in `LooplimitsContainSymbolicVariable[]` is set if any of the loop limits contain symbolic variables. Similarly `LooplimitsAreLinear[]` specifies whether the loop limits are linear. This type of memorization enables the dependence tests to determine the nature of the loop limits.

The subscript expressions of the given pair of array references are saved in `acoeff` and `bcoeff`. The symbolic variables that may present in the subscript expressions can be invariant or variant with respect to the loop-nest. The nature of the symbolic variables in the subscript expressions is determined by the routine `processSymbolicVariable()`. If the variable is constant with respect to the given loop-nest, the corresponding subscript expression is linear. Otherwise the subscript expression is non-linear. This information is saved in the vector `SubscriptContainSymbolicVariable[]`. A call to `call_dep_tests()` invokes each dependence test in the order specified by the user.

Since all the tests are conservative in nature, if any one of the test determines independence, no edge is added to the dependence graph. The conservative nature of the dependence tests may permit dependence directions for which no dependence exists. The other way is not possible. Hence, the intersection of the set of direction vectors returned by various tests is taken. Only these direction vectors are added to the dependence graph.

Once dependence is assured between any two statements, the type of dependence is to be determined . The following algorithm determines the type of dependence based upon the lexical positions of the references and direction vector of the dependence.

Algorithm: Type of Dependence

```
Input: Direction vector and types of references.
Output: Type of dependence.
typeofdependence()
{
    for (each element in the direction vector)
        if (element = '<') then {
            if (both are write references) then
                return output dependence from $stat_1$ to $stat_2$;
            else
                return flow dependence from $stat_1$ to $stat_2$;
            endif
        }
        elif (element = '>') then {
            inverse the direction vector;
            if (both are write references) then
                return output dependence from $stat_2$ to $stat_1$;
            else
                return antidependence from $stat_1$ to $stat_2$;
            endif
        }
    if ($stat_1$ and $stat_2$ are same) then
        return no dependence;
    elif ($stat_2$ lexically precede $stat_1$)then
        if (both are write references)then
            return output dependence from $stat_2$ to $stat_1$;
        else
            return anti dependence from $stat_2$ to $stat_1$;
        endif
    elif (both are write references)then
        return output dependence from $stat_1$ to $stat_2$;
    else
        return flow dependence from $stat_1$ to $stat_2$;
}
```

Chapter 3

Tests for Dependence Analysis

3.1 Introduction

The data dependence tests are decision algorithms which determine the existence of an integer solution to a given set of linear diophantine equations. As mentioned in Section 2.4, this is an NP-complete problem. Most of the dependence tests check for some necessary conditions for the solutions to exist. Some other tests employ integer programming techniques to find a general solution to the given dependence problem.

The dependence tests are conservative in nature. That is, they assume dependence unless it is explicitly ruled out by a violation of the necessary conditions. The various tests differ in the methods they employ and the amount of information they provide. Some tests give only "yes" or "no" answer to a given dependence problem. At the other extreme some tests can enumerate all the solutions.

3.2 Various properties of dependence tests

- Some tests, for example Gcd and Banerjee's test consider each subscript at a time. These tests may fail when the given subscripts are coupled[1] because

[1]If the same index variable appears in more than one subscript expressions, the latter are known as coupled subscripts

the solution for one subscript expression may not satisfy the other subscript expressions.

- The various tests differ in how accurately they can determine the dependence. For example, the Power and Omega tests are more accurate than the simple tests - Gcd and Banerjee's test[13]. The more accurate tests, obviously, employ more expensive techniques to solve the diophantine equations.

- The iteration space formed by the loop bounds can be triangular, rectangular or trapezoidal in form. The constant loop bounds result in rectangular regions. On the other hand, if the bounds themselves are functions of the enclosing loop index variables, the iteration space may be triangular or trapezoidal. The rectangular iteration spaces represent the simplest cases and some tests, for example, the I test and Lambda test are applicable only for such spaces.

- Most of the dependence tests are not applicable when unknown variables occur in either loop limits or in subscript expressions. The I test can be applied even when some of the loop bounds are not known. The Omega and Power tests can be applied to the dependence problem with symbolic constants.

- The single index exact test given in [17] handles subscript expressions with one loop index variable. It is an exact test and can also be used to enumerate all the solutions within the loop bounds.

- It is more useful for the purpose of restructuring to know the direction or distance vector of the dependence, if it exists. But some tests can provide only "yes" or "no" answer and hence, are not much useful.

Unfortunately there is no general test which handles all the cases efficiently. An extensive analysis on the performance of various dependence tests is reported in [16]

3.3 I Test

The I test is an inexact subscript by subscript test proposed by Kong et. al [8]. The I test combined Gcd and Banerjee's test in the sense that it determines integer solutions within the bounds. Moreover, it may determine independence even if some of the loop limits are not known.

Definition 3.1 Let $a_1, a_2, \ldots, a_n$, L and U be integers. The equation,

$$a_1 I_1 + a_2 I_2 + \ldots + a_n I_n = [L, U] \tag{3.1}$$

where $M_k \leq I_k \leq N_k$ $(1 \leq k \leq n)$, is referred to as an *interval* equation, will be used to denote the set of ordinary equations consisting of

$$\begin{aligned} a_1 I_1 + a_2 I_2 + \ldots + a_n I_n &= L \\ a_1 I_1 + a_2 I_2 + \ldots + a_n I_n &= L + 1 \\ \vdots \quad &\quad \vdots \\ a_1 I_1 + a_2 I_2 + \ldots + a_n I_n &= U \end{aligned}$$

The I test is based on the following theorem

Theorem 3.1 *Let $a_1, a_2, \ldots, a_n$ be integers. For each k, $(1 \leq k \leq n-1)$, let each of M_k and N_k be either an integer or the distinguished symbol '*'(not known), where $M_k \leq N_k$ if both M_k and N_k are integers. Let M_n and N_n be integers, $M_n \leq N_n$. If $|a_n| \leq U - L + 1$, then the interval equation is $(M_1, N_1; M_2, N_2; \ldots; M_n, N_n)$-integer solvable iff the interval equation*

$$a_1 I_1 + a_2 I_2 + \ldots + a_{n-1} I_{n-1} = [L - a_n^+ N_n + a_n^- M_n, U - a_n^+ + a_n^- N_n]$$

is $(M_1, N_1; M_2, N_2; \ldots; M_{n-1}, N_{n-1})$-integer solvable.

The algorithm is discussed in more detail in[8]

3.4 Lambda Test

3.4.1 Description

The Lambda test is an inexact test which is devised to handle coupled subscripts. Usually, subscript by subscript tests determine dependence by examining each dimension of the array references. If the examination of any dimension shows no solution then there exists no data dependence between the two references. However, the sets of solutions for any two dimensions may be disjoint. The subscript by subscript tests fail to recognize such cases frequently. The Lambda test is more effective in such cases [6]. This test can be applied only when the array references have constant loop bounds. However, trapezoidal regions can be converted to rectangular regions so that Lambda test can be applied. This may lead to inaccuracy, but the result is still conservative. One more limitation of this test is that it does not handle unknown variables in the subscript expressions.

Formally coupled subscripts are described as follows.

Definition 3.2 We denote the set of loop indices in *reference₁* and *reference₂* by $IND = \{i_1, i_2, \ldots, i_{l_1}, j_1, j_2, \ldots, j_{l_2}\}$ and denote the index set of $reference_1$ and $reference_2$ that appear in the array dimension j, $1 \leq j \leq m$, by $IND_j \equiv \{i | i \in IND$ and i appears in either f_j or $g_j\}$

1. If $IND_{d_1} \cap IND_{d_2} \neq \emptyset$, then dimension d_1 and d_2 are said to be coupled and $reference_1$ and $reference_2$ are said have coupled subscripts.

2. If dimension d_1 and d_2 are coupled, and d_2 and d_3 are coupled then d_1 and d_3 are also coupled.

The Lambda test can be divided into two parts.

1. finding the linear combination of coupled subscripts

2. finding maximum and minimum values of the linear combination using loop bounds and direction vectors

Finding linear combination: The loop bounds and the given dependence directions correspond to a bounded convex set **V** in R^n. Each linear equation in Equation 2.3 is a hyperplane π in R^n space. The intersection **S** of m-hyperplanes corresponds to the common solutions to all the equations. If **S** is empty then there is no data dependence. If the hyperplane π intersects **V** then the corresponding equation has real valued solutions within the loop bounds. Any subscript by subscript test can test for this condition. But to determine the real valued simultaneous solutions, it is needed to determine whether **S** itself intersects **V**. If any of the hyperplanes do not intersect **V**, then **S** cannot intersect **V**. However, even if every plane in Equation 2.3 intersects **V**, it is still possible that **S** and **V** are disjoint.

Theorem 3.2 $\mathbf{S} \cap \mathbf{V} = \emptyset$ *iff there exists a hyperplane, π, which corresponds to a linear combination of equations in Equation 2.3, $\sum_{i=1}^{m}(\lambda_i \vec{a_i}\vec{v}) + \sum_{i=1}^{m} \lambda_i c_i = 0$, such that $\pi \cap \mathbf{V} = \emptyset$.*

The first part of the Lambda test finds the necessary and sufficient λ-tuples to form the linear combination of m equations. We first consider two coupled subscripts and then expand the concept to generalized version.

Two coupled subscripts: An arbitrary linear combination of two equations in Equation 2.3 can be written as $\lambda_1 f_1 + \lambda_2 f_2 = 0$. The domain of λ_1, λ_2 is the whole two dimensional space. Let

$$\begin{aligned} f_{\lambda_1,\lambda_2} &\equiv \lambda_1 f_1 + \lambda_2 f_2 \\ &\equiv (\lambda_1 a_{11} + \lambda_2 a_{21})v^{(1)} + (\lambda_1 a_{12} + \lambda_2 a_{22})v^{(2)} + \ldots + (\lambda_1 a_{1n} + \lambda_2 a_{2n})v^{(n)} \end{aligned}$$

f_{λ_1,λ_2} can be viewed as a linear function of (λ_1, λ_2) in two dimensional space with $v^{(1)}, v^{(2)} \ldots, v^{(n)}$ fixed. The coefficient of each $v^{(i)}$ in f_{λ_1,λ_2} is a linear function of (λ_1, λ_2) in two dimensions. i.e., $\psi^{(i)} \equiv \lambda_1 a_{1i} + \lambda_2 a_{2i}$. This is a straight line, called ψ line, passing through the origin and makes the space into two halves. There are atmost n ψ lines which together divide the space into atmost $2n$ regions. Each region is a cone called λ cone.

Lemma 3.1 *Suppose* V *is defined by loop bounds but not by dependence directions. If* $f_{\lambda_1,\lambda_2} = 0$ *intersects* V *for every* (λ_1, λ_2) *in every* ψ *line then* $f_{\lambda_1,\lambda_2} = 0$ *also intersects* V *for every* (λ_1, λ_2) *in* R^2.

Intuitively, the Lemma 3.1 states that if all the λ-tuples on the boundary of λ cone make f_{λ_1,λ_2} intersect V then $f_{\lambda_1,\lambda_2} = 0$ intersects for all (λ_1, λ_2). According to Lemma 3.1, we get infinite number of λ-tuples, which is not feasible for practical purposes. For each λ_1, λ_2 we get different f_{λ_1,λ_2}. The maximum and minimum value of f_{λ_1,λ_2} depends upon the sign of the coefficients of that function. Now consider V defined by direction vectors as well as loop bounds. Let v_i and v_j be the same loop indices occurring in two references. Direction vectors gives us the relation between two variables v_i and v_j. The maximum and minimum value of each variable depends on the direction vector as well as coefficients a_i and a_j of v_i and v_j respectively [10]. Let $\phi^{i,j} = \lambda_1(a_{1i} + a_{1j}) + \lambda_2(a_2^{(i)} + a_2^{(j)})$, since v^i and v^j are related by direction vectors. This is a line called ϕ line, in two dimensional space. Now the minimum value and maximum value of the function f_{λ_1,λ_2} depends not only on the sign of the coefficients of each v but also on the sign of $\phi^{(i,j)}$. There are atmost $n/2$ ϕ lines. All ϕ lines and ψ lines divide two dimensional space into 3n regions. It is proved that if $f_{\lambda_1,\lambda_2} = 0$ intersects V for any (λ_1, λ_2) in every ψ line and ϕ line, then $f_{\lambda_1,\lambda_2} = 0$ also intersects V for every (λ_1, λ_2) in a ψ line or ϕ line. Hence it is suffice to test a single point in each line. The algorithm can be summarized as follows

1. find a point on each ψ line (or ϕ line).

2. form the linear combination of equations in Equation 2.3

3. if the resultant equation intersects V then goto Step 1. otherwise report no dependence.

Generalized version: An arbitrary linear combination of m equations can be written as

$$\begin{aligned} f_{\lambda_1,\lambda_2,\ldots,\lambda_m} &\equiv \lambda_1 f_1 + \lambda_2 f_2 + \ldots + \lambda_m f_m = 0, \\ &\equiv (\Sigma_{j=1}^m \lambda_j a_j^{(1)})v_1 + (\Sigma_{j=1}^m \lambda_j a_j^{(2)})v_{(2)} + \ldots + (\Sigma_{j=1}^m \lambda_j a_j^n)v_{(n)}. \end{aligned}$$

The Lemma 3.1 can be expanded to find sufficient number of λ tuples to determine whether $f_{\lambda_1,\lambda_2,\ldots,\lambda_m} = 0$ intersects **V** in R^m space for arbitrary $(\lambda_1, \lambda_2, \ldots, \lambda_m)$. The sufficient number of λ tuples is any one point on each λ cone boundary. In m-dimensional space the λ-cone is formed by any $m-1$ ψ or ϕ planes. Hence, there is a finite set of hyperplanes in R^n such that **S** intersects **V** if and only if every hyperplane in the set intersects **V**. If **V** is defined by loop bounds alone, then there are no more than $\binom{n}{m-1}$ hyperplanes in the set. On the other hand, if **V** is defined by loop bounds as well as dependence directions, there are no more than $\binom{3n/2}{m-1}$ hyperplanes in the set. Algorithm to find out sufficient λ-tuples is simple.

1. Generate $\binom{n}{m-1}$(or $\binom{3n/2}{m-1}$)combinations of integers from 1 ... n.
2. Use the elements in each combination as an index to form $m-1$ simultaneous equations out of n (or 3n/2) equations.
3. Solve m-1 simultaneous equations for a λ tuple.
4. Form the linear combination of Equation 2.3 using the λ tuple.
5. If the resultant function intersects V goto 1. Otherwise no dependence.

Does S intersects V ?: The maximum and minimum values of the resultant linear combination can be found by using Banerjee's test. However, Li et al. proposed set of rules to find the minimum value and maximum value of the function depending on the direction vector. Either of the methods can be used. If maximum is less than minimum then there is no dependence.

3.4.2 Implementation Details

The routine `convertTrapezoidalRegionToRectangularRegion()` is used to convert trapezoidal region to rectangular region. `call_lambda_test()` determines the number of coupled subscripts and initiate appropriate routine based on the number of coupled subscripts. To find the maximum value and the minimum value of the linear combination, the routine `doesSintersectV()` uses the set of rules in [10].

3.5 Omega Test

3.5.1 Description

Omega test is an exact test, which uses Fourier-Motzkin elimination method to solve set of inequalities formulated from the dependence problem. Omega test consists of three parts. Formulation of equality and inequality constraints comprises the first part. Applying Fourier-Motzkin elimination method on the set of inequalities is another part. This in turn produces the system of inequalities in terms of loop index variables. Finding out the direction and distance vectors is the last part.

Formulation of problem: The input to the Omega test is a set of linear equalities $(\Sigma_{0\leq i\leq n} a_i x_i = 0)$ and inequalities $(\Sigma_{0\leq i\leq n} a_i x_i \geq 0)$, where $x_0 = 1$, a_0 is the constant term and **V** is the set of loop indices being manipulated. Each constraint is normalized. A normalized constraint is one in which all the coefficients are integers and gcd of the coefficients is 1. Given a problem involving equality and inequality constraints, we convert all equality constraints to inequality constraints. The resultant set of inequalities has integer solutions if and only if the original problem had integer solutions. *Euclid*'s generalized algorithm discussed in the Section 3.6 can be used for this purpose, since equality constraints arises because of the subscript expressions of the array reference. However, the following approach is followed by Omega test to eliminate equality constraints for better performance. To eliminate the equality $\sum_{i \in V} a_i x_i = 0$,

1. Check if there exists a $j \neq 0$ such that $|a_j| = 1$. If so, we eliminate the constraint by solving for x_j and substituting the result into other constraints.

2. Otherwise, let k be the index of the variable with the coefficient that has the smallest absolute value ($k \neq 0$) and let $m = |a_k| + 1$. $\widehat{mod}$ operation is defined as $a \widehat{mod} b = a - b\lfloor(a/b + 1/2)\rfloor$. We create a new variable σ and produce the constraint $m\sigma = \sum_{i \in V}(a_i \widehat{mod} m)x_i$.We solve this constraint for x_k.

$$x_k = -Sign(a_k)m\sigma + \textstyle\sum_{i \in (V-\{k\})} Sign(a_k)(a_i \widehat{mod} m)x_i$$

In the original constraint, this substitution produces:

$$-|a_k|m\sigma + \sum_{i \in V-\{k\}} ((a_i - (a_i \widehat{mod} m)) + m(a_i \widehat{mod} m))x_i = 0$$

Normalize this constraint. goto 1.

Inequality constraints are processed to

- find contradictory constraints, in which case there is no dependence to the system
- eliminate redundant constraints
- tighten the constraints

Intuitively, we are decreasing the solution space constituted by the dependence problem in the cartesian coordinate system. If the problem involves atmost one variable and has passed the above tests, we report that it has integer solutions.

Fourier-Motzkin elimination: In the second part of the Omega test we apply Fourier-Motzkin elimination to eliminate a variable from the set of inequalities. Intuitively, Fourier-Motzkin variable elimination finds the $n-1$ dimensional object[2] in n dimensional space. There may be integer points in the shadow of an object, even if the object itself contains no integer points. This is called real shadow. To determine real shadow, consider two constraints on z:a lower bound $\beta \leq az$ and an upper bound $az \leq \alpha$(where a and b are positive integers and α and β are linear combinations of the remaining variables in the system of inequalities). We combine these constraints to get $a\beta \leq abz \leq b\alpha$. The constraint $a\beta \leq b\alpha$ is the shadow of intersection of these two constraints. By combining the shadow of the intersection of each pair of upper and lower bounds on z we obtain, what is called real shadow. We define dark shadow of the object, which ensures that for every integer point in the dark shadow, there is an integer point in the object above it. To determine the dark shadow, consider the case in which there is an integer solution to $a\beta \leq b\alpha$. The dark shadow is $(b\alpha - a\beta) > (a-1)(b-1)$.

[2]solution space defined by the set of constraints

The algorithm for checking for the existence of integer solutions to a set of constraints is summarized as follows:

1. choose the 'best' possible variable to eliminate. 'Best' variable must be able to provide either exact projection, where dark and real shadows are identical, or the coefficients of the corresponding variable as close to zero as possible.

2. Calculate real and dark shadows of the set of constraints. If both the shadows are equal then there are integer solutions to the original set of constraints, iff there are integer solutions to the shadow.

3. Otherwise, if there are no integer solutions to the real shadow, no solution to the system of inequalities. If there are integer solutions to the dark shadow, system has solutions.

4. Otherwise determine the largest coefficient a of z in any upper bound on z. For each lower bound $bz \leq \beta$, test if there are integer solutions to the original problem combined with $bz = \beta + i$ for each i such that $(ab - a - b)/a \geq i \geq 0$.

Direction and distance vectors: For each common loop, a new variable is introduced. The value of this variable determines the distance and sign of the variable determines the direction of dependence w.r.t the loop. Now the problem is projected on to these variables. Unprotect the variable whose sign is determined or the variable is uncoupled [3]. Now the entire problem is projected onto the remaining variables. Otherwise, choose one protected variable and generate subproblems for two or three possible signs for the variable. This process is done recursively to enumerate all the vectors. This is the only dependence test that does not use the frame work that is discussed in Section 2.5 to generate all the possible direction vectors.

[3]an uncoupled variable does not associate with any other variable in any inequality

3.5.2 Implementation Details

`initialize_EQ_n_GEQ()` forms the problem from the array reference subscript expressions and loop limits. `initializeOmega()` initializes the internal data structures used by the Omega test. `simplifyProblem()` accepts a pointer to the problem and returns 1 if a solution exists, otherwise 0. The problem is projected on to the protected variables and the resultant problem is returned. `unprotectVariable()` accepts a pointer to a problem, a variable and unprotects it. `constraintVariable()` constraints the variable to have the sign +1, 0, -1, unprotects it, reduces the problem and returns 1 if solution exists. `calculateDDVectors()` calculates direction vectors and distance vectors by the method discussed above.

3.6 Power Test

3.6.1 Description

The Power test is a combination of *Euclid*'s generalized algorithm and Fourier-Motzkin elimination method. *Euclid*'s algorithm is used to determine whether the simultaneous linear diophantine equations, derived from subscript expressions of array references, has a solution without considering the loop bounds. Fourier-Motzkin method is used to eliminate variables in a system of inequalities derived from the loop limits and direction vectors. The byproduct of *Euclid*'s algorithm is the system of one or more linear equations to which simultaneous diophantine equations are reduced.

Consider the set of simultaneous diophantine equations in Equation 2.3. The coefficient matrix **A** is

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{bmatrix}$$

Algorithm 5.5.1 in [3] transforms the coefficient matrix $\mathbf{A}_{n \times m}$, formed from diophantine equations, into unimodular integer matrix $\mathbf{U}_{n \times n}$ and echelon integer $\mathbf{D}_{n \times m}$ such that $\mathbf{UA} = \mathbf{D}$. Elementary row transformations are applied on $\mathbf{A}$ to transform it into an echelon matrix $\mathbf{D}$. The same sequence of operations are performed on the unit matrix $\mathbf{I}_{n \times n}$ yielding $\mathbf{U}$, where n is the number of variables in the problem of equations. The echelon matrix $\mathbf{D}$ and unimodular integer matrix $\mathbf{U}$ are of the form

$$\mathbf{D} = \begin{bmatrix} d_{11} & d_{21} & \dots & d_{m1} \\ 0 & d_{22} & \dots & d_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & d_{mm} \\ \vdots & \vdots & \ddots & \vdots \end{bmatrix}$$

$$\mathbf{U} = \begin{bmatrix} u_{11} & u_{21} & \dots & u_{n1} \\ u_{12} & u_{22} & \dots & u_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ u_{1n} & u_{2n} & \dots & u_{nn} \end{bmatrix}$$

If there is an integer solution vector $\mathbf{t}$ such that $\mathbf{tD} = \mathbf{C}$, then $\mathbf{hA} = \mathbf{C}$, where $\mathbf{C}$ is the constant vector of simultaneous equations and h is loop indices vector. After finding $\mathbf{D}$ and $\mathbf{U}$, the test finds values for t_1 through t_m by solving $\mathbf{t}\ \mathbf{D} = \mathbf{C}$ using a simple back propagation algorithm. If there are feasible solutions, the extended GCD algorithm stops here and assumes dependence. It also gives formulas that can be used to specify the index variables h_1, h_2, ..., h_n in terms of the 'free' variables t_{m+1}, t_{m+2}, ..., t_n derived from the matrix product $\mathbf{h} = \mathbf{t}\ \mathbf{U}$. If the dependence system has constant dependence distances, we can find them by subtracting corresponding equations. That is, the dependence distance for a loop at depth $k(1 \leq k \leq c)$ is found by subtracting the equations for i_k and j_k. Suppose that $i_k \equiv h_{2k-1}$ and $j \equiv h_{2k}$;the two equations are subtracted by looking at

$$\mathbf{tU}_{*,2k-1} - \mathbf{tU}_{*,2k}$$

(where $\mathbf{U}_{*,x}$ is a column of the the matrix U). If the dependence distance is fixed, this will have non zero coefficients only for t_1 through t_m, which were previously solved. If there are non zero coefficients for any other t_v, where $v > m$ the dependence distance is not constant.

Till now we have exploited all the capabilities of *Euclid*'s algorithm. In the above phase we haven't considered all the constraints on the dependence system such as loop bounds. Loop limits and direction vector information comprises a set of linear inequalities or constraints on the set of 'free' variables.

The Power test constructs a list of upper and lower bounds on each 'free' variable t_k, each lower and upper bound will be linear combination of t_{m+1}, t_{m+2}, ..., t_{k-1}.These give the boundaries to the solution space to the dependence equation;if the solution space is nonempty, then the dependence equation has solutions that satisfy all the conditions. Each lower bound for t_k will be of the form

$$lb_k t_k \geq lb_0 + lb_{m+1}t_{m+1} + \ldots + lb_{k-1}t_{k-1}, lb_k > 0 \tag{3.2}$$

These bounds are derived from the constraints on the index variables, such as the loop limits. Each index variable h_i is defined by the extended GCD algorithm as some linear combination of the 'free' variables. In addition, the upper and lower limits of each index are themselves linear combinations of outer loop index variables. Thus the constraint $h_i \geq l_i$ can be algebraically reduced to an inequality constraint on one of the 'free' variables, of the form Equation 3.2.

After formulating the inequalities based upon the loop limits, direction vectors are generated according to the frame work discussed in Chapter 2. For each direction vector element we get another inequality which derives a lower or upper bound on one of the free variables.

Given a set of inequalities in terms of 'free' variables, Fourier-Motzkin visits each 'free' variable (from t_n down to t_{m+1}) comparing each lower bound to each upper bound. Each comparison will be of the form

$$(lb_0 + lb_1t_{k-1} + \ldots + lb_{k-1}t_{k-1})/lb_k \leq \quad t_k \quad \leq (ub_0 + ub_1t_1 + \ldots + ub_{k-1}t_{k-1})/ub_k$$

from which we derive

$$(lb_kub_0 - ub_klb_0) + (lb_kub_1 - ub_klb_1)t_1 + \ldots + (lb_kub_{k-1} - ub_klb_{k-1})t_{k-1} \quad \geq \quad 0$$

If any of the coefficients are non zero, this will derive another lower or upper limit on another lower numbered 'free' variable. If all the coefficients are zero, then we have simple inequality $lb_k ub_0 - ub_k lb_0 \geq 0$. If this inequality is not satisfied, then there is a solution to the dependence system, and thus no dependence.

3.6.2 Implementation Details

`dd_sim_T()` is a routine to invoke Power test. It returns distance and direction vectors. `dd_reduce()` applies generalized GCD algorithm to reduces the matrix A to unit matrix at the same time determining unimodular matrix. `dd_solve()` solves for t_1, t_2, ..., t_s using back substitution method. `dd_fixed()` computes distance vector if any distance vector exists.`dd_T_init()` initializes bounds for each 'free' variable. `dd_enforce_limit()` finds out all the possible vectors using Fourier-Motzkin elimination method. The variable `dd` is cleared if no solution is detected in any of the above steps.

Chapter 4

Restructurer

4.1 Introduction

The goal of automatic parallelization is to transform sequential code to parallel code. The restructuring of the given program must not affect the semantics of the program. The restructured loop-nest can be executed on different processors in parallel. If there is a dependence relation in the loop-nest that prevents parallelization, then restructuring compiler can attempt several simple transformations on the loop-nest to remove the dependence relation. For example, when the dependence graph of the loop is acyclic, then statement reordering will always allow parallelization. A topological sort of the dependence graph specifies how to reorder the statements. If a cycle in the graph cannot be reduced to a single statement, then loop distribution can be applied to remove the cycle to a separate loop.

The loop restructuring techniques use dependence information to determine their feasibility and profitability. The feasibility study of each transformation determines the applicability of the transformation for a given loop-nest. However, the profitability study of the transformation determines whether transformation extracts any parallelism from the code. Once a restructuring technique is chosen to be feasible and profitable, appropriate modifications are performed on the program representation of the loop-nest. Hence the implementation of a restructuring technique contains three parts.

1. Feasibility study

2. Determining profitability

3. Changes to the underlying program representation

Every transformation has its own feasibility and profitability criteria. At the backend of restructurer, the program representation of the loop-nest is converted back to control flow graph[1]. In the following sections we discuss certain restructuring techniques that are incorporated into FRAMES.

4.2 Program Dependence Graph

The control flow graph (CFG) has been the usual representation for the control flow relationships of a program. But this representation does not allow restructurer to determine the control conditions of an operation readily. A statement Y is said to be control dependent on X according to the following definition.

Definition 4.1 Let G be a control flow graph. X and Y be nodes in G. Y is control dependent on X if and only if

1. There exists a directed path P from X to Y with any Z in P (excluding X and Y) post dominated by Y and

2. X is not post-dominated by Y.

If Y is control dependent on X then X must have two exits; following one of the exits from X always results in Y being executed; while taking the other exit may result in Y not being executed. Condition 1 can be satisfied by a path consisting of a single edge. Condition 2 is always satisfied when X and Y are the same node.

Many of the transformations required to change the control dependences of statements in a given program. For example, new control statements are added to CFG in cycle shrinking, the order of control statements is changed in loop interchange and so on. To support these types of operations, an efficient representation of the given program is needed. The essential properties of such a representation are to

- make the control dependences of every operation in a given program explicit.[1]
- make the operations on the representation transparent

The Program Dependence Graph(PDG) is the program representation used in FRAMES. Since PDG connects computationally relevant parts of the program, many code improving transformations require less time to perform than with other program representations. A single walk of these dependences is sufficient to perform many optimizations. For details of PDG, reader is advised to refer to[4].

4.2.1 Description of PDG

PDG consists of three types of nodes.

Predicate nodes correspond to the conditional statements in a given program.

Region nodes summarize the set of control conditions for a node and also group all nodes having the same set of control conditions together.

Statement nodes correspond to the imperative statements in a given program.

The predicate nodes and region nodes are connected by a conditional edge and named *True* or *False*. Region nodes and statement nodes are connected by an unconditional edge. Each predicate has a unique successor for each truth value. The strongly connected component (SCC) in the program dependence graph contain nodes consisting of predicates that determine an exit from the loop. The other nodes in the PDG not in the SCC lie on some path of control dependence edges from a node in the SCC. Intuitively, these correspond to the body of the loop. Nested loops appear as distinct SCCs with a control dependence edge between the outer loop and each immediate inner loop. Loops at the same level appear as SCCs with a common ancestor region. Consider the following loop-nest.

[1]It is not possible to determine exactly under what control conditions an operation in a given program is performed

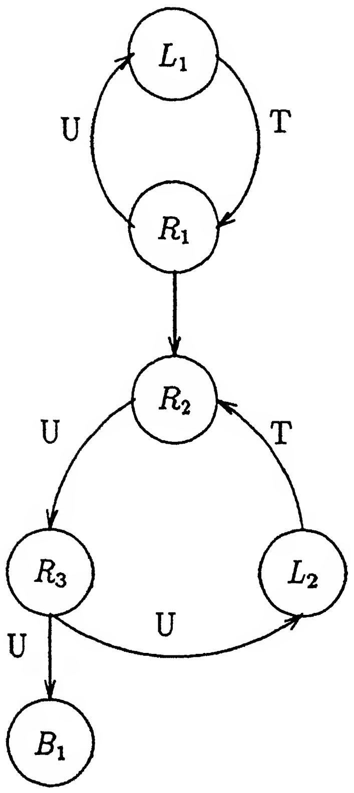

Figure 4.1: The Program dependence graph

```
N = 50
DO I = 1, N
  DO J = 1, N
    S1A(I, J) = B(I-1, J)
    S2B(I, J) = A(I-2, J)
  ENDDO
ENDDO
```

The PDG for the above loop-nest is given in Figure 4.1.

4.3 Parallel code generation

Once the data dependence graph is constructed, we proceed to generate parallel code for the given loop-nest as follows.

1. Find all the strongly connected components(SCC) in the data dependence graph.

2. Reduce the DDG to an acyclic graph by treating each SCC as a single node.

3. Apply node-splitting, cycle shrinking and loop interchange on each SCC.

4. Generate code for each SCC in an order consistent with the dependences. That is, by using a method similar to topological sort, first generate code for nodes depend on no others, then for nodes that depends only on blocks for which code has already been generated, etc.

The code generation procedure can generate loop-nests for the SCCs in a given DDG. The loop-nests may consist parallel loops depending on the dependence relations between the nodes in that particular SCC. However, it need not give up on an SCC that does not allow parallel loops. Restructuring compiler can attempt several transformations on the SCC. These transformations may be intended to remove cycles or to extract any inherent parallelism in the given SCC. For example, node-splitting is used to remove dependence cycles and cycle shrinking is used to extract inherent parallelism from a given SCC. Hence the algorithm given in [2] is modified to incorporate the third step. The algorithm to generate loop-nest for a given SCC is reproduced from [2]. The loop-nest generated may be a mix of parallel loops and serial loops.

Algorithm: Loop-nest Generation for SCCs

```
Input: Acyclic data dependence graph.
Output: CFG of the resulting loop nest.
parallelcode()
{
    init_sccq(sccs, noofsccs, sccq, inedgeno);
    while((scc = pop_q(sccq)) != NULL)
    {
        for(depth = 1 to maxdepth of the loop-nest) {
            let $D_{depth}$ be the dependence graph consisting of all
            dependence edges in DDG which are at level-depth or greater
            and which are internal to scc;
            if(dependence cycle exsits in $D_{depth}$)
                generate a level-depth DO statement;
            else
                generate a level-depth DOALL statement;
        }
        sort statements in scc in topological order;
        for(each statement in scc)
            insert_stmt();
        process_sccq(sccs, noofsccs, i, sccq, inedgeno);
    }
}
```

The routines `init_sccq()` and `process_sccq()` are to ensure that the parallel code generated for SCCs is in topological order. `init_sccq()` calculates the number of incoming edges of each SCC from other SCCs. All SCCs with inedge number zero are enqueued into sccq. These SCCs are not dependent on any other SCCs. Hence code can be generated for these SCCs without data dependence violations. `process_sccq()` recalculates the inedge numbers of each SCCs, not taking the edges from SCC that is passed as an argument into consideration. It also enqueues SCCs with inedge number zero. insert_stmt() is the routine which takes a statement and the control conditions under which the statement is executed and inserts the statement in CFG. This is the basic routine to construct CFG from PDG. The algorithm to convert PDG from CFG given in [4] is not suitable for our purpose [15]. The algorithm for insert_stmt() and the proof of the algorithm is given in [15]

4.4 Node Splitting

Loop parallelization is impossible when statements in the body of a given loop are involved in a dependence cycle [12]. Dependence cycles that involve only *flow* dependences are usually hard to break. There are cases, however, where dependence cycles can be broken resulting in total or partial parallelization of the corresponding loops. The cycle breaking may be possible if dependence cycles involve *flow* and *anti/output* dependences. *Anti* and *output* dependences are false dependences and can either be ignored or eliminated. The backward anti/output dependence edges are ignored. The forward anti/output dependence edges are removed by node splitting transformation. The following example illustrates the node splitting transformation.

```
DO I = 1, N
    $S_1$:A(I) = B(I)
    $S_2$:B(I-1) = A(I+1)
ENDDO
```

Before Node splitting

```
DO I = 1, N
   $S_2$': TEMP(I) = A(I+1)
ENDDO
DO I = 1, N
   $S_1$: A(I) = B(I)
      $S_2$: B(I-1) = TEMP(I)
ENDDO
```

After Node splitting

If the dependence cycle in SCC is not broken as the result of node splitting then

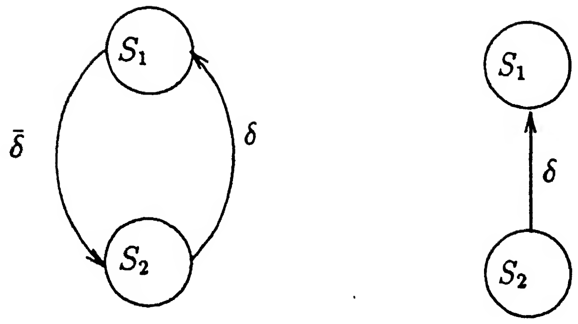

Figure 4.2: DDG before and after Node Splitting

the transformation is not profitable.

Definition 4.2 : A subgraph $G'(V, E')$ is called restricted graph of data dependence graph $G(V, E)$, if $E' \subseteq E$ and every *flow* dependence edge in E is also present in E'.

If dependence cycle is present in G', where G' is the restricted graph of a given SCC, it is not possible to break the dependence cycle by means of node splitting. Hence the transformation is not profitable.

Algorithm: Node Splitting

Input: Data dependence graph of a given loop-nest.
Output: Data dependence graph

```
callnodesplit()
{
    find SCC's in a given DDG.
    for (each SCC in the DDG) {
        if (SCC contains more than one statement) then {
            construct restricted graph $G'$ of SCC.
            if (cycle is not present in $G'$) then
                remove each anti/output dependende
                forward edge by node splitting
        }
    }
}
```

The node splitting is not affected by the presence of scalar dependences. The transformation does not change the control dependences of any of the statements. The new assignment statements generated during node-splitting are also control dependent on the same conditions as that of the node that is split.

4.5 Loop Interchange

The loop interchange [17, 11] is the process of switching loops in a given loop-nest. This is one of the most powerful restructuring techniques. This technique is mainly architecture dependent. Some of the architecture dependent optimizations [17] are

- multi-processor machines perform better with parallel outer loops
- most of the parallel machines give better performance with larger loop limits
- vector machines work well with large vectors than smaller vectors
- multi processors perform better when many parallel operations are possible

The loop interchange can be used to switch the loops in the best possible way depending on the underlying architecture of the machine, thereby achieving good performance. We consider interchange of simple loops in this work. Interchange of triangular loops as well as some advanced loop interchanges are discussed in [17].

4.5.1 Feasibility of Loop Interchange

All loop interchanges will not yield correct results. The requirements for two simple loops L' and L'' to be interchanged are specified as follows

1. Loop L'' is nested perfectly within L'
2. Loop limits of L'' are invariant of L' index
3. There are no statements S_u and S_w in L'' with a dependence relation $S_u\delta_{(<,>)}S_w$.

The proof of the last requirement is given below.

Theorem 4.1 *Suppose $S_u\delta_{(<,>)}S_w$. Then there are values i_1, i_2, and j_1, j_2, where $i_1 < j_1$, $i_2 > j_2$ and $S_u[i_1, i_2]\ \delta\ S_w[j_1, j_2]$. If the loops are interchanged, then $S_w[j_1, j_2]$ will be executed before $S_u[i_1, i_2]$ since $j_2 < i_2$. That is sink will be executed before source, thus violating the data dependence relation. Suppose there are no dependence relations with $(<, >)$ direction vectors. The only possible dependence direction vectors between statements S_u and S_w in the loop are $(<, <)$, $(<, =)$, $(=, <)$, $(=, =)$. Now, suppose there is a dependence relation $S_u\delta(<, <)S_w$. Then there are values i_1, i_2 and j_1, j_2 where $i_1 < j_1$, $i_2 < j_2$ and $S_u[i_1, i_2]\delta S_w[j_1, j_2]$.*
If the loops are interchanged, then $S_u[i_1, i_2]$ will still be executed before $S_w[j_1, j_2]$, thus satisfying the data dependence relations. A similar argument holds for the direction vectors $(<, =)$, $(=, <)$, $(=, =)$.

4.5.2 Profitability of Loop Interchange

Profitability of loop interchange differs from machine architecture to machine architecture. This fact makes loop interchange the most higher level transformation that a compiler can perform. Compiling for multiprocessors need parallel loops. Better performance is achieved if the parallel loops are the outer loops in a given loop-nest. Loop interchange can be used to switch serial outer loop and a parallel inner loop.

```
DO I = 1, N
   DOALL J = 1, M
      S1:  A(I, J) = B(I, J)*C(I, J) + A(I+1, J)
   ENDDO
ENDDO
```

In the above loop-nest, the scheduler [12] assigns a processor for every iteration of parallel loop. This assignment performed N number of times corresponding to N number of serial outer loop iterations. Obviously, the execution of parallel loop iterations must be synchronized N number of times. The number of fork and join operations to synchronize parallel loop iterations is prohibitively high. This may result in parallelized code that is worse than the sequential code. This type of profitability can be captured by the depth of dependence. The interchange of two loops is said to be profitable if the depth of dependence between any two statements in a given loop-nest is increased prior to interchange. The algorithm for loop interchange is similar to bubble sort algorithm and is given below.

Algorithm: Loop Interchange

Input: SCC and the loops in a given loop-nest.
Output: The best order of loops in terms of profitability and feasibility.

```
loopinterchange()
{
    find perfectly nested loops in the given loop-nest;
    let the loops $l_1, l_2, \ldots, l_n$ are perfectly nested loops;
    for ($L_1 = l_{n-1}$ to $l_1$)
        for ($L_2 = l_n$ to $L_{1+1}$)
            if (switching of $L_1$ and $L_2$ is feasible and profitable) {
                interchange the loops $L_1$ and $L_2$;
                interchange the corresponding elements of
                the direction vectors in DDG;
                modify the PDG;
            }
}
```

The application of loop interchange on the PDG given in Figure 4.1 transforms to the representation given in Figure 4.3. The loop L_3 is the duplicate of L_2 and the loop L_4 is the duplicate of L_1.

4.6 Cycle Shrinking

In many of the cases it is not possible to eliminate a cycle in the dependence graph. Usually, this occurs when the cycle is formed by flow dependence edges. The cycle shrinking can be applied to exploit the inherent parallelism in a serial loop or in a nest of serial loops. As mentioned in Chapter 2, distance vector explicitly specifies the number of iterations between two successive memory location references. Obviously, those iterations can be executed in parallel. The cycle shrinking cannot be applied with dependence distance one. There are basically two cycle shrinkings

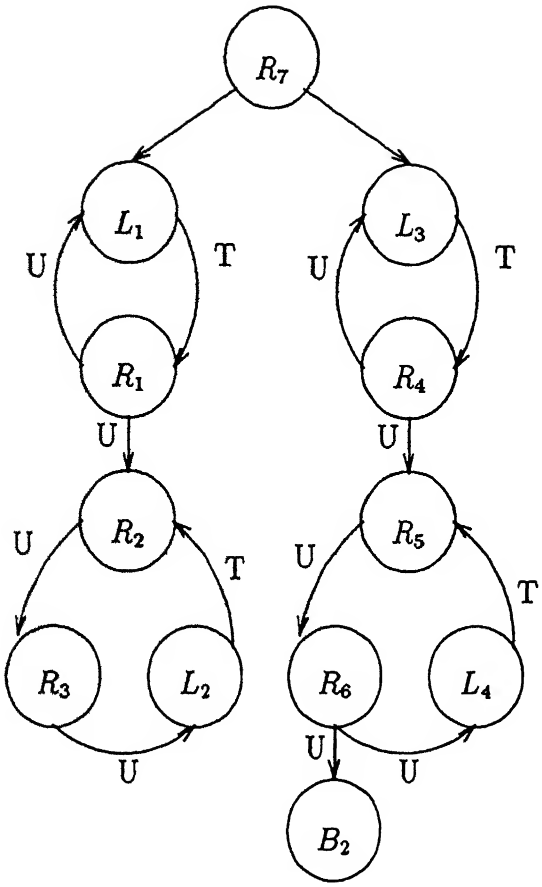

Figure 4.3: PDG after loop interchange

techniques based on the complexity of the loops. Their performance depends on the number of iterations, that can be executed in parallel. The number iterations that can be executed in parallel is referred to as reduction factor λ.

4.6.1 Calculation of Reduction factor

Singly nested loops: Let us consider there are n statements S_1, S_2, ..., S_n, in a loop-nest that are involved in a dependence cycle. Moreover, the dependence distance between S_1 and S_2 is k_1, between S_2 and S_3 is k_2, and so on until k_n. We consider the following cases.

- All the dependence distances are same and constant. Then the reduction factor, λ, is any one of distances k_1, k_2, $\cdots$, k_n (for further details and proofs, reader may refer to [12]).

- All the dependence distances are constant, but distances between different dependences are different. The subscript expressions of the form aI+b and aI+c give rise to such dependence distances. The reduction factor, λ, is given by

$$\lambda = min\{k_1, k_2, \ldots, k_n\}$$

- when the distances vary with different instances of dependence, λ is calculated by the following formula.

$$\lambda = min_{1 \leq i \leq n}\{\phi(\delta_i)\}$$

  where $\phi()$ generates different instances of dependence δ_i. In singly nested loops this happens when we have array subscripts of the form aI+b where $a \geq 1$ or $a \leq -1$.

It is evident that the type of reduction factor depends upon the subscript expressions in the array references.

Theorem 4.2 *Consider a DO loop with k statements which are all involved in a dependence cycle. If the reduction factor of the cycle is λ, then cycle shrinking increases the speed up of the loop by a factor of $\lambda * k$.*

Multiple nested loops: There are two versions of cycle shrinking that can be used for multiply nested loops. They differ in the way reduction factor is computed.

- *True dependence(TD) shrinking*: True distance of each dependence can be calculated from the distance vector. Let $(d_1, d_2, \ldots, d_n)$ be the distance vector of a dependence δ. The true distance t is

  $$t = \sum_{i=1}^{s} d_i \prod_{j=i+1}^{s} (U_j - L_j + 1)$$

  where s is the number of common loops enclosing the references. Let ($t_1, t_2, \ldots, t_n$) be the true distances of n dependences involved in a cycle, the reduction factor is

  $$\lambda = min(t_1, t_2, \ldots, t_n)$$

  In TD shrinking multi-dimensional array space is treated as a linear space. The loop limits must be known in order to compute the true distances. The values of the distances *per se* are not needed, but determining the minimum true distance in a cycle is essential for the TD shrinking. The following example demonstrates this transformation.

```
DO I = $L_1, U_1$
  DO J = $L_2, U_2$
    A(I, J) = B(I-3, J-5)
    B(I, J) = B(I-2, J-4)
  ENDDO
ENDDO
```

```
DO K = 1, $(U_1 - L_1 + 1)(U_2 - L_2 + 1)$, $\lambda$
  $T_1$ = $(KDIV(U_2 - L_2 + 1)) + L_1$
  $T_2$ = $((K + \lambda)DIV(U_2 - L_2 + 1)) + L_1$
  $T_3$ = $K\%(U_2 - L_2 + 1) + L_2 - 1$
  $T_4$ = $((K + \lambda)\%(U_2 - L_2 + 1)) + L_2 - 2$
  DOALL J = $T_3$, M
    A($T_1, J$) = B($T_1 - 3, J - 5$)
    B($T_1, J$) = B($T_1 - 2, J - 4$)
  ENDDO
DOALL I = $T_1 + 1, T_2 - 1$
  DOALL J = $L_2, U_2$
    A(I, J)=B(I-3, J-5)
    B(I, J)=B(I-2, J-4)
  ENDDO
ENDDO
DOALL J = $L_2, T_4$
  A($T_2, J$) = B($T_2 - 3, J - 5$)
  B($T_2, J$) = B($T_2 - 2, J - 4$)
ENDDO
ENDDO
```

T values for a loop nest having n normalized loops can be calculated by the formulas.

$$\left.\begin{aligned} T_{l_i} &= ((k)div(\textstyle\prod_{j=i+1}^{n} U_j)) + 1 \\ T_{u_i} &= ((k+\lambda)div\textstyle\prod_{j=i+1}^{n} U_j) + 1 \end{aligned}\right\} for \;\; 1 \leq i < n$$

$$\left.\begin{aligned} T_{l_n} &= (kmodU_n) \\ T_{u_n} &= ((k+\lambda)MODN_n) - 1 \end{aligned}\right\} for \;\; i = n$$

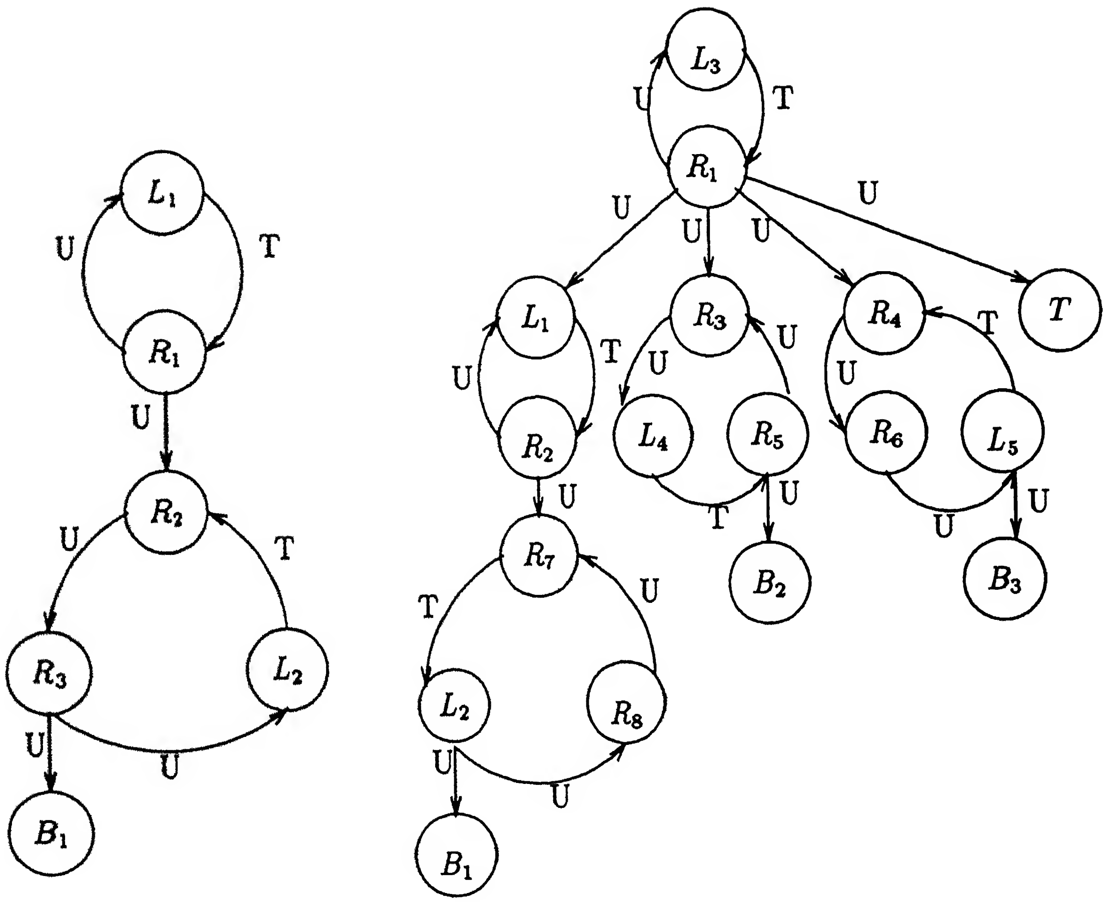

Figure 4.4: The PDG before and after application of TD shrinking

- *Selective shrinking*: The dependence cycle within k nested loops can be viewed as k different dependence cycles, one for each individual loop. Each dependence in a cycle is labeled with the corresponding element of its distance vector. Selective shrinking computes the reduction factor λ_i, $(i = 1, 2, \ldots, k)$ for each loop in the nest starting with the outer most loop. The process stops when for some j, $(1 \leq j \leq k)$, $\lambda_j \geq 1$. Then the j^{th} loop in the loop nest is blocked by a factor of λ_j. In addition, all the loops nested inside the j^{th} loop are transformed to DOALLs. The following loop-nest illustrates this method.

```
DO I = L1,U1
  DO J = L2,U2
    A(I, J) = B(I-1, J-4)
    B(I, J) = A(I-2, J-5)
  ENDDO
ENDDO
```

Loop nest

```
DO I = L1,U1
  DO K = L2,U2,4
    DOALL J = K,K+3
      A(I, J) = B(I-1, J-4)
      B(I, J) = A(I-2, J-5)
    ENDDO
  ENDDO
ENDDO
```

Transformed loop

Algorithm: Cycle Shrinking

```
Input: SCC and loops enclosing SCC
Output: Transformed code
cycleshrinking(SCC)
{
    find the feasibility of cycle shrinking.
    determine type of shrinking and reduction factor.
    if (type of shrinking is selective shrinking)
    {
        block the corresponding loop by reduction factor
        convert all the inner loops to parallel loops
        make changes to PDG
    }
    else
    {
        /* True dependence shrinking */
        produce outer most loop with step, reduction factor
        produce first and second unrolled loops
        make replicate of strongly connected components
        replace loop variables by T variables and new loop variable
        make changes to PDG
    }
}
```

4.7 Loop Fusion

Traditionally, loop fusion is used to increase the size of the loop body, thereby reducing the loop overhead. In parallel processors this one of the main goal, since scheduling of a loop onto various processors is very expensive [12].

In loop fusion, two loop bodies are combined to make a single loop. The two loops must be at the same depth, called the *depth of fusion*. Moreover, the loops must satisfy the following conditions

1. Both the loops must be of same type; i.e., both are either doall loops or do loops.

2. The loop limits and steps of both the loops must be same.

3. Both the loops must be executed under similar control conditions.

4. No exit out of either of the loops is permitted.

5. There should not be any cross iteration dependence at the depth of fusion.

If two loops satisfy above mentioned criteria, then they can be fused. Consider the following loop-nest.

```
DO I = 1, N
   S1:  A(I) = B(I)+C(I)
ENDDO
DO I = 1, N
   S2:  D(I) = A(I)*2
ENDDO
```

```
DO I = 1, N
S1:  A(I) = B(I) + C(I)
S2:  D(I) = A(I)*2
   ENDDO
```

This loop-nest satisfies the above mentioned conditions and hence, it can be fused to a single loop. The effect of Loop fusion on PDG is shown in Figure 4.5.

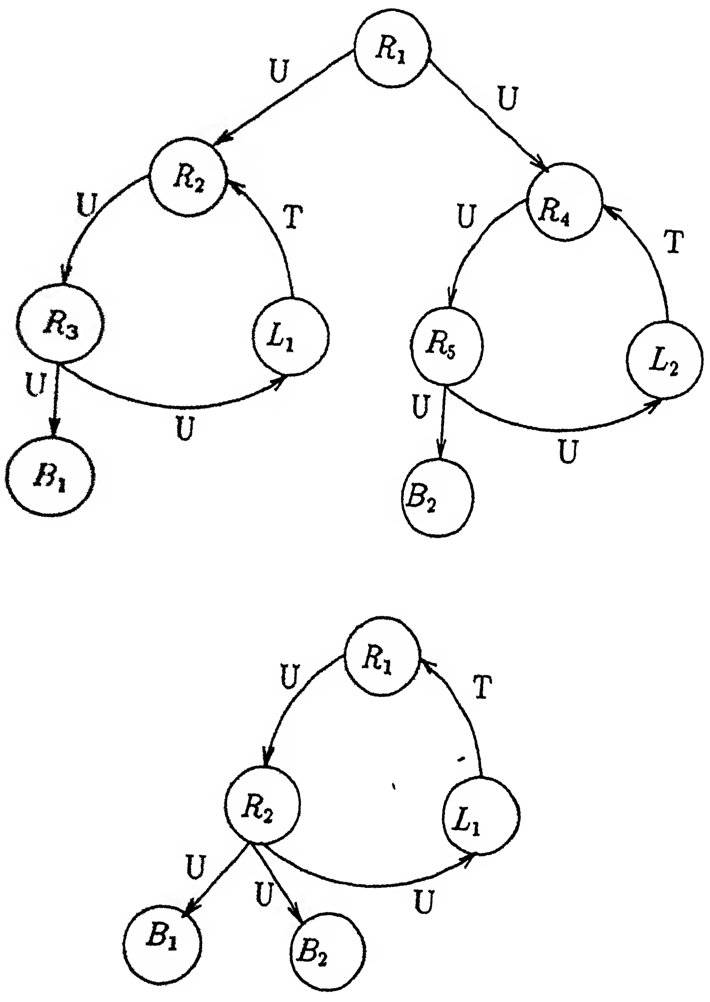

Figure 4.5: The effect of L [illegible])G

Algorithm: Loop Fusion

```
Input: set of SCCs
Output: set of SCCs
loopFusion(sccs, noofsccs, depth)
{
    init_sccq(sccs, noofsccs, sccq, inedgeno);
    while((workingelement = pop_q(sccq)) != NULL) {
        enqueue(tempq2, workingelement);
        while((testingelement = pop_q(sccq)) != NULL) {
            if(workingelement and testingelement are fuse able)then {
                make modification to the PDG
                fuse testingelement with workingelement.
            }
            else{
                enqueue(tempq1, sccq);
                process_scc(sccs, noofsccs, testingelement, sccq);
            }
        }
        /* Reenqueue all the elements that were not fused to sccq */
        while((element = pop_q(tempq1)) != NULL)
            enqueue(sccq(element));
        remove data dependence edges with depth of dependence
        more than 'depth';
        initialize ownsccs with the elements in workingelement.
        loopFusion(ownsccs, noofownsccs, depth + 1);
        while((element = pop_q(tempq2)) != NULL)
            enqueue(sccq(element));
    }
}
```

The Gcd and Banerjee tests are applied to determine cross over dependence at the depth of fusion. If the dependence exists the two SCCs are not fusable. `tempq1` is used to store SCCs that are not fused to the workingelement. `tempq2` is used to store the SCCs that are fused together with the working element. Once loop fusion is applied at a given depth of fusion, the routine is called recursively at `depth+1`.

Chapter 5

Epilogue

5.1 Testing

Till now we have been discussing different aspects of data dependence analysis, data dependence tests and various restructuring techniques and their implementation. Now we discuss testing that FRAMES has under gone.

5.1.1 Test Programs

FRAMES has been tested on various bench-mark programs extracted from standard packages like *LINPACK*, Livermore Kernel. Moreover, many test programs had been written to test various aspects of implementation. These programs contain basically various types of loop nests. Loops in loop-nests vary from one to four. Care has been taken to include all types of array references such as coupled subscripts, subscript with single loop index and subscript with multiple loop indices. The number of dimensions of array references vary from one dimension to three dimensions. Some of the references contain symbolic variables in their subscripts. Still some bias is present in these written programs. While testing for restructuring we never considered loops with a scalar variable definition and use within the loop nest. IF statments with in the loop-nests are rare.

5.1.2 Output of FRAMES

The output of data dependence analysis is a data dependence graph. The graph is printed in the following format: Each vertex and list of edges originating from that vertex. Result of each dependence test for each pair of array reference is written into file *frames.log* created in current directory. This will enable programmers to evaluate the performance of various dependence tests. Output of FRAMES is restructured code of the given program. The ensuing subsections manifest the capabilities of FRAMES. Each section is meant for a specific ability of the restructuring compiler.

5.1.3 An Example for DDA

The following example shows the effect of introducing sophisticated tests. Traditional Banerjee's and GCD test reported dependence for the array reference *a*. This clearly shows their inability to handle coupled subscripts. The Lambda and the Omega test reported independence which enabled the restructurer to apply Loop distribution.

Input Program

```
c     Capability of Lambda and Omega test
      dimension a(100, 100, 100), b(100)
      do i = 1, 10
          do j = 1, 20
              a(2*i+3*j+10, 3*i+j+9, i+j) = b(i, j)
              b(j, i) = a(i-j+11,2*i-j+7, i+j)
          enddo
      enddo
      stop
      end
```

Output of FRAMES:

```
*** Restructured program ::  ***
c         Capability of Lambda and Omega test
          dim a(100,100,100), b(100,100)
          forall i = 1, 10
              forall j = 1, 20
                  a(2*i+3*j+10,3*i+j+9,i+j) = b(i,j)
              endfor
          endfor
          forall i = 1, 10
              forall j = 1, 20
                  b(j,i) = a(i-j+11,2*i-j+7,i+j)
              endfor
          endfor
          stop
          end
```

5.1.4 An Example of Loop interchange

The following example illustrates the loop distribution as well as loop interchange.

Input Program:

```
c      Loop interchange and Loop distribution
       dimension a(100, 100), b(100, 100), c(100, 100)
       do j = 1, 100
           do i = 1, 100
               a(i, j) = b(i, j-1)
               b(i, j) = a(i, j-1)
               c(i, j) = a(i, j-1)
           enddo
       enddo
       stop
       end
```

Output of FRAMES

```
*** Restructured program ::  ***
c        Loop interchange and Loop distribution
         dim a(100,100), b(100,100), c(100,100)
         forall i=1,100
             do j=1,100
                 b(i,j)=a(i,j-1)
                 a(i,j)=b(i,j-1)
             enddo
         endfor
         forall j=1,100
             forall i=1,100
                 c(i,j)=a(i,j-2)
             endfor
         endfor
         stop
         end
```

The first two statements in the input program form a cycle in the DDG. Hence

Output of FRAMES

```
*** Restructured program ::  ***
c         Loop Fusion
          dim a(100, 100), b(100, 100, 100)
          dim e(100, 100), c(100, 100), d(100, 100, 100)
          forall i = 1, 100
              forall j = 1, 100
                  a(i, j) = 5.0
                  c(i, j) = e(i, j)
                  forall k = 1, 100
                      b(i, j, k) = 10.0
                      d(i, j, k) = 15.0
                  endfor
              endfor
          endfor
          stop
          end
```

There is no data dependence between the statements in the loop-nest. Hence the DDG contains only vertices without edges. Adjacent loops are fused to a single loop since there is no cross over dependences.

5.1.6 A Nightmare

The following example demonstrates the inability of FRAMES to extract the full amount of parallelism from the code. This is basically due to the ordering of transformations.

Input Program:

```
c      A nightmare of FRAMES
       dimension a(100,100), b(100,100), c(100,100)
       read(*,*) n
       do i = 1, n
           do j = 1, n
               a(i, j) = b(i-4, j)
               b(i, j) = a(i-5, j) + c(i, j)
           enddo
       enddo
       stop
       end
```

There exists a dependence cycle between the two statements. The output of frames is given below.

Output of FRAMES:

```
c      A nightmare of FRAMES
       dim a(100,100), b(100,100), c(100,100)
       read(*,*)n
       do j = 1, n
           do i_nvar_1 = 1, n, 4
               forall i = i_nvar_1, i_nvar_1+3
                   a(i,j) = b(i-4,j)
                   b(i,j) = a(i-5,j) + c(i,j)
               endfor
           enddo
       enddo
       stop
       enddo
```

According to FRAMES, the Loop interchange is applied before cycle shrinking. In the above example, Loop interchange is justified because the depth of dependence is increased as a result of loop interchange. The cycle shrinking could able to block the inner loop. If cycle shrinking is applied before loop interchange then outer loop is blocked, so that maximum number of iterations can be executed in parallel.

5.2 Conclusions

We have enhanced the data dependence analysis capabilities of FRAMES by providing a frame work which takes results from dependence tests and builds a data dependence graph. The I, the Lambda, the Omega and the Power tests are implemented for this purpose. These new tests handle all types of subscript expressions [16] that appear in scientific programs. Restructuring capabilities of FRAMES are enhanced by incorporating loop interchange loop fusion and cycle shrinking.

5.3 Future Developments

Many more data dependence tests can be incorporated. For example, GA test [16] enumerates tha solutions for diophantine equations. This is of great help in extracting the fine grain parallelism from the code. Some of the dependence tests potential has not been fully exploited. For example, Omega test gives predicate expression in terms of symbolic variables whose values are not known at compile time. This information can be coded into the program. If the predicate is evaluated to truth value *false* then there is no dependence and the loop-nest can be executed in parallel.

Data dependence across loop-nests is not determined. Let us consider loop nests LN_1, LN_2 and LN_3 in lexical order in a given program. Assume there exists dependence from LN_1 to LN_3. And LN_2 is not dependent on any other loop-nest. In this case LN_1 and LN_2 can be executed in parallel if dependence across loop-nests is determined. This will certainly enhances the speed-up.

The loop interchange is implemented only for perfectly nested loops with constant loop bounds. This can be further extended to imperfectly nested loops as well as loop bounds defining trapezoidal region in iteration space. The goal of the present implementation is to move parallel inner loops to outer loops. There are many other goals, for example, making array references in the loop-nest unit stride which can be achieved by loop interchange. Many more restructuring techniques can be implemented. Main limitations of restructuring phase of FRAMES can be summarized as follows.

- The inability to handle scalar dependences with in the loop-nest.
- Restructuring techniques are ordered;Node splitting, Loop interchange, Cycle shrinking and Loopfusion. This order cannot be altered even if the program demands different ordering to produce efficient code.

Better program representation which unifies control dependences and data dependences is still a research topic. This will enable us to handle scalar dependences and control statements within the loop-nest more efficiently. An algorithm which determines dynamically the best ordering of various transformations depending on the given program is yet to be developed.

References

[1] A. V. Aho, Sethi R., and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] R. Allen and K. Kennedy. Atomatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[3] U. Banerjee. *Dependence Analysis for Super Computing*. Kluwer Academic Publishers, 1988.

[4] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3), JULY 1987.

[5] S. Gautam. Fortran-D parallelizing compiler: Scheduling phase. Master's thesis, I. I. T., Kanpur, 1993.

[6] G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. In *ACM SIGPLAN '91 Conf. Programming Language design and Implementation*, pages 15–29, Toronto, Canada, June 1991.

[7] S. Hiranandani, K. Kennedy, and T. Tseng. Compiling Fortran D for MIMD distributed memory machines. *Communications of ACM*, 35(8):66–80, August 1992.

[8] X. Kong, D. Klappholz, and K. Psarris. The I test: A new test for subscript data dependence. In *Proceedings of the 1990 International Conference on Parallel Processing*, St. Charles, IL, August 1990.

[9] Sanjeev Kumar. Structure of frames. Technical report, I. I. T. Kanpur, 1993.

[10] Z. Li, P.-C. Yew, and C.-Q. Zhu. An efficient data dependence analysis for parallelizing compilers. *IEEE Trans. on Parallel and Distributed Systems*, 1(1):26–34, 1990.

[11] D. A. Padua. Advanced compiler optimizations for supercomputers. *Communications of ACM*, 29(12):1184–1200, December 1986.

[12] C. D. Polychronopolous. *Parallel Programming and Compilers.* Kluwer Academic Publishers, 1988.

[13] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, 1992.

[14] Ananda R. Fortran-D parallelizing compiler: Restructuring phase. Master's thesis, I. I. T., Kanpur, 1993.

[15] R. K. Singh. Fortran-D parallelizing compiler: Front end phase. Master's thesis, I. I. T., Kanpur, 1993.

[16] S. Singhai. To be published. Master's thesis, I. I. T., Kanpur, 1995.

[17] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.

[18] M. Wolfe and C.-W. Tseng. The power test for data dependence. *IEEE Trans. on Parallel and Distributed Systems*, 3(5):591–601, 1992.